Informatics in Education, 2009, Vol. 8, No. 2, 261–280

© 2009 *Institute of Mathematics and Informatics, Vilnius*

Verificator: Educational Tool for Learning Programming

Danijel RADOŠEVIĆ, Tihomir OREHOVAČKI, Alen LOVRENČIĆ
Faculty of Organization and Informatics, University of Zagreb
Pavlinska 2, 42000 Varaždin, Croatia
e-mail: {danijel.radosevic, tihomir.orehovacki, alen.lovrencic}@foi.hr

Received: July 2009

Abstract. The paper introduces Verificator, our learning programming interface aimed for learning programming in C++ at the university beginners' level. In teaching programming some specific problems concerning the teaching itself as well as the organization of the teaching process need to be considered. One of the biggest problems is that students tend to adopt certain bad programming habits in their attempt to more easily deal with their examinations, such as trying to write programs without any syntax and logical checking. It is very hard to help them correct those errors once they are deeply rooted. Our students' web questionnaire and its results show that the majority of problems in learning programming among our students arise from the gap between the understanding of programming language syntax and problem-solving algorithms. Verificator prevents students from making a lot of errors they are likely to make in learning programming and helps them to learn programming language syntax and adopt good programming habits.

Keywords: learning programming, teaching software, Verificator.

1. Introduction

The main goal of this paper is to introduce Verificator, our software tool aimed at helping students to more easily acquire programming skills, and improving the process of teaching programming at the university beginners' level. Over the years, while teaching programming at the University of Zagreb, Faculty of Organization and Informatics in Varaždin, Croatia, we have faced some problems inherent in the teaching itself as well as those related to the organization of the teaching process. Students tend to adopt some bad programming habits in their attempt to pass the exam. There were cases in which students would learn big fragments of program code by rote, without any or with hardly any understanding. There were also students who wrote large program code without any syntax and logical testing, which produced a huge number of errors, discouraging the students from programming altogether.

Our students' web-form questionnaire was conducted in 2009 on a population of 182 respondents (students at the University of Zagreb, Faculty of Organization and Informatics, at the beginning of the spring semester, in the Programming I course). Some students had foreknowledge in programming from the Informatics 1 course (basics of Python), and some of them, from the secondary school and individual work.

The questions included students' previous skills in programming and informatics in general, together with their current problems in acquiring programming skills. Students declared a lack of foreknowledge and being distracted by other duties in their studies to be the most common problems. However, some of the most severe problems, like the fear of programming, are mostly correlated to a lack of programming practice as well as the misunderstanding of basic programming concepts and programming language (in our case, C++) syntax.

Therefore we introduced Verificator in our teaching process as a software tool which should motivate our students towards taking the right approach to acquiring programming skills by:

- preventing students from copying programs or parts of programs from their colleagues. All programs have to be typed (not copied or loaded) in Verificator, and contain the author's data, together with the checksum (which verifies that the program is really written in Verificator),
- pushing students to check their programs' syntax after adding several lines of code. That prevents students from writing programs by rote and collecting a huge number of errors,
- helping students to find syntax and logical errors in their programs. There are tools for syntax analysis of programs (e.g., using of curly brackets and open program structures) and debugging tools (e.g., using breakpoints).

Verificator works as a programming interface (editor with programming tools) which uses a standard C++ compiler (we used a *gcc* freeware compiler).

2. Related Work

Programming is one of the fundamental skills to be adopted by students of information sciences at the end of the first year of their study. It is also expected that they will know basic programming concepts and have developed the algorithmic approach to solving problems. Basic programming concepts include the knowledge of control structures (sequence, selection, and iteration), mechanisms of aggregation (array, structure, union), pointers, functions, etc. When teaching programming, our aim is to teach novice students the basic principles of using programming concepts regardless of the programming language. However, goals are sometimes hard to meet in the real-life context. The research which was conducted at the multi-national level has shown that novice students, after completing and passing their courses in which they learn the basic concepts of programming, still encounter problems creating simple program solutions (Lister *et al*., 2004). This is probably accounted for by the fact that novice programmers do not have enough experience or skills regarding which programming constructs to use or where and how to use them. Furthermore, students are often faced with syntax errors. It is common for a small set of identical mistakes to occur regularly with all novice students (Spohrer and Soloway, 1986; Ben-Ari, 1998; McCracken *et al*., 2001). In order to help students in their first steps in learning programming, many visualization and animation software systems are developed.

Visualization tools can be very useful in teaching programming, primarily because their main purpose is to facilitate students' understanding of code execution by guiding them through a series of animated techniques (Mulholland, 1998; Hundhausen *et al*., 2002). In addition, the results of several studies (Fowler *et al*., 2000; Cardellini, 2002, Thomas *et al*., 2002) showed that the majority of students (more than 75%) adopt new content best when it is presented visually. Thus, the use of a visualization tool in teaching programming allows students to create a mental model of program execution or its parts (Tudoreanu, 2003). However, Naps *et al*. (2002) emphasizes that visualization tools can be useful only if students are actively involved in the learning process. Owing to the above-mentioned advantages of visualization tools in the educational process, they have been actively used in the study and animation of algorithms (Garner, 2003; Costelloe, 2004) and data structures (Grissom *et al*., 2003; Laakso *et al*., 2005). Visualization tools can be divided into two groups: flowcharts and simulation tools.

Flowcharts are usually a good choice of a visualization tool for teaching programming, primarily because they can present the algorithm as a mental model that a student with little or no prior knowledge can easily understand. By using flowcharts, students will easily perceive the program flow and later, when they try to independently solve problems, this can be of great benefit. However, whereas flowcharts can be useful for visualization and modeling of simple programming concepts, they are not suitable for large and complex applications (Scott *et al*., 2006). From among the set of visual programming environments based on a flowchart primarily intended for teaching programming, the following should be pointed out: Rapid Algorithmic Prototyping Tool for Ordered Reasoning (RAPTOR; Carlisle *et al*., 2004), Flow Chart Interpreter (FCI; Atanasova and Hristova, 2003) and Flint (Crews, 2001). All the mentioned tools enable students to create algorithms through a combination of basic graphical symbols. In addition, studies have shown that the average grades of students using visualization development environments based on a flowchart were significantly higher than those who learned programming by using traditional development environments (Crews, 2001; Carlisle *et al*., 2004).

The second group consists of visualization simulation tools. The general approach to animation of algorithms defined in procedural higher level languages began with the BALSA system (Brown and Sedgewick, 1984) after which TANGO (Stasko, 1990) and Polka (Stasko and Kraemer, 1993) systems were developed at the Georgia Institute of Technology. Their main purpose was the teaching of dynamic behavior of complex algorithms. However, those systems were not very successful, primarily because their animation design was more appropriate for experts in the field than for novice students. At the University of Helsinki, Jorma Tarhio and Erkki Sutinen developed Jeliot (Haajanen *et al*., 1997), which was focused on the animation of programs. The results of the empirical experiment correlated with the use of the Jeliot tool in teaching programming have shown that the students' motivation for learning increased, but also that its interface is too complicated and confusing to novice students (Latte *et al*., 2000). Therefore, Jeliot 2000 was developed and exclusively designed for beginners. Jeliot 2000 is experimentally used for teaching programming in high schools, where the results of a survey showed that students who used Jeliot 2000 software better understand programming concepts and have a better

developed vocabulary than the control group (Ben-Bassat Levy *et al*., 2001). The latest version of Jeliot is Jeliot 3, developed at the University of Joensuu. Its primary purpose is monitoring the performance of Java code, and thus, easier understanding of control structures and finding errors in the code (Kannusmäki *et al*., 2004). One of the most successful visualization simulation software packages is certainly Karel, The Robot (Pattis, 1981). It is a tool in which a student controls a robot in a virtual microworld with four basic actions and thus learns basic programming constructs. Karel is the software that facilitates novice students to learn the Pascal programming language and has been used for teaching programming in schools and universities for years. With the development of object-oriented paradigm, changes in the programming languages which were used in teaching programming also occurred. Therefore it is not surprising that Karel has had several different versions: Karel++ (C++ version; Bergin *et al*., 1997) and Guido van Robot (Python version; Elkner, 2004) but none of them has achieved the same success as their original. It is assumed that the Alice programming environment is the main successor of Karel. It is a 3-D Interactive Graphics Programming Environment that allows scripting and modeling prototype objects that a virtual simulation world is made of (Pausch *et al*., 1995). In simulations, students can use simple scripts to control the appearance and behavior of objects. During script execution, visualization allows students to establish correlation between the program and the animated action of the complex and thus understand how basic programming constructs operate. TRAKLA2 is a visual environment that allows the assessment of simulation of algorithms and data structures (Korhonen *et al.,* 2003). VILLE is a visual tool that supports Java, C ++ and pseudo programming languages (Kaila *et al*., 2009). Programming examples can be simultaneously displayed in two programming languages, and thus show different implementations of programming concepts. In addition, VILLE allows monitoring of the execution of the program and thus of the result of changes in the output arising from changes in the value of variables. Finally, we should by all means mention BlueJ as an example of a static visualization tool whose characteristics are a directly parameterized call of the method and automatic generation of applet skeleton. Its basic purpose is acquiring object-oriented concepts in the Java programming language (Kölling *et al*., 2003).

From all the aforementioned examples it can be concluded that visualization tools are very useful in teaching programming to novice students, primarily because they can show and explain programming concepts in a very simple way. However, their main disadvantage is that the majority of them are focused on only one programming language and the simplest program constructs. Furthermore, the aim of teaching programming is that students understand basic programming concepts and that they become able to apply those concepts during the problem solving process, regardless of implemented programming language. Unfortunately, most of these tools are too focused on the visualization component, and less on learning the syntax and semantics of the selected programming language. Therefore, we believe that the visualization tools are more appropriate for teaching programming in elementary schools, but not at universities where the student needs to learn algorithmic approach to solve given problems. Finally, the results of research that has followed the use of most of the aforementioned tools have not revealed any significantly better results and students still experience the same difficulties when learning basic

programming (Milne and Rowe, 2002; Lahtinen *et al*., 2005; Butler and Morgan, 2007). Therefore we decided to develop Verificator, an environment in which students will learn strategies they will use in solving problems and developing the necessary concepts and skills to create computer programs.

3. Our Approach to Teaching Programming

There are several problems that we have to deal with in the process of teaching programming. The most important one is a huge disproportion in the students' foreknowledge. Programming is an obligatory basic course that students take in the second semester. Their previous knowledge and techniques highly depend on their previous education, which varies from school to school. For many of them the course is their first encounter with programming, so they do not have any experience doing it. On the other hand, there are quite a few students with moderate, or even good programming experience. This problem can be traced back to the organization of programming courses in primary and secondary school. In primary school, which has the same curriculum in the whole country, informatics is not an obligatory subject, although most pupils take it. Even so, these courses teach only the basics of computer usage, and do not deal with any programming. On the other hand, best pupils take part in programming competitions and acquire a moderate knowledge of programming techniques and programming languages Basic and Logo. Most secondary school informatics courses are optional as well.

To make all students interested in an obligatory programming course, it is necessary to equalize their previous knowledge as much as possible. To do that we introduced an optional zero level programming course in which students are not awarded any ECTS credits. It was introduced as a free of charge winter school of programming. However, the number of attendants turned out to be a problem. Namely, more than 80% of our students attended this winter school. As a result, owing to such a high percentage of students interested in the preliminary programming school, we decided to change the way of organizing it and include it in the obligatory basic course of Informatics held in the first semester.

The second dilemma we had to solve was the choice of the programming language to be used in the programming course. The language to be used has to be widely acknowledged and used in practice, easy to learn and needs to contain all the important concepts of programming languages. As we are talking about a basic programming course, it is obvious that any functional or logic programming language will be excluded. These languages require an understanding and usage of advanced programming strategies, such as recursion and backtracking method, and are therefore not appropriate for a basic course. The three main procedural language lines nowadays are Basic-like, Pascal-like and C-like languages. There is a significant difference between these three lines. While C-like and Pascal-like languages are more frequently used by professionals, languages from the Basic-like language line do not contain many important language concepts and are more appropriate for informative level courses. Furthermore, we deliver our programming course to two different types of students. The first of them are students enrolled in

the undergraduate university study of information and business systems. Although there are two distinct study programs at this level –information systems and business systems – their first two semesters are identical, so the students in both programs have to be treated as a single group. The second group is students enrolled in the vocational study of information technology usage in business systems. The students in the latter group have to get only informative knowledge of programming, so Visual Basic is used in their course. The first group is more interesting because the programming knowledge they acquire is important for several courses they attend further on in the study. The knowledge of programming is thus crucial for their future education as well as for their careers after graduation. Consequently, the approach to this group of students should be more systematic and serious. Therefore, Visual Basic is not a good choice for them. In choosing between C-like and Pascal-like languages we were guided by three different criteria:

- the popularity of the language in practice,
- the availability of all important programming concepts,
- popular OS platform support.

Unfortunately, there is no data about the usage of programming languages in Europe, so we used U.S. data, namely, the TIOBE Programming Community Index of programming language usage. A great dominance of C-like languages in current practiceis shown in (TIOBE, 2009) and (Lovrenčić *et al.,* 2009). Four main C-like languages (C, C++, Java and C#) take up more than 50% of the programming code in USA. On the other hand, three languages from the Pascal-like line of languages (Delphi, PL/SQL and Pascal) accounted for less than 4%. In this situation, the C-like line of languages was an easy choice. The narrowing of one language was a slightly harder task. It is obvious that Java, as the most widely-used language, was a very serious candidate. On the other hand, Java has some drawbacks that make it inappropriate for the course. The first problem with Java is the fact that it requires Java Virtual Machine. It is an advanced concept that makes Java the most portable language, although it also makes the basic concepts of compiling and interpreting of the code more difficult to explain. The second problem with Java is garbage collection, another advanced concept that makes Java hard to understand for beginners. When it comes to C#, its popularity is relatively small and decreasing, although, with its C syntax and Pascal semantics, it would be the best choice. Another problem with C# is its strong connection to MS Windows, and the absence of serious open-source compilers or interpreters for it. So, C and C++ languages remain candidates. We decided to choose C++, regardless of its smaller popularity in comparison with C, because it is object-oriented, which makes Java concepts easier to teach in the aforementioned courses.

After choosing the programming language, we had to organize the course. In previous years the main organizational problem had been the poor connection between the lectures giving theoretical knowledge and the laboratory exercises, which are meant to provide 'hands-on' experience. The solution we devised was the 3-level organization: the lectures that give theoretical knowledge, but also introduce concepts of C++ programming language, the auditory exercises that demonstrate the usage of concepts introduced in the lectures, and syllabus exercises in which individual work and skill sharpening is ensured.

Continuous learning plays a major role in a programming course. The reason for that is the fact that the understanding of subsequent concepts that are taught largely depends on the understanding of those previously taught and any discontinuity results in problems understanding and following the lectures. This also explains why we abandoned the classical concept of examination through mid-term exams and final exam, and introduced a new system of 10 blitz-tests and 10 syllabus exercises that constitute the final grade for each student. Blitz-tests follow the concepts that are taught during the lectures, while syllabus exercises are used for grading of practical skills that students acquire through auditory exercises.

The main presumption of our examination system is individual work. The dilemma that arises is whether the assignments in syllabus exercises should be disclosed to students in advance. If the exercises are not disclosed in advance, students have to create the concept, algorithm and program during the test. That requires assigning a less difficult task, so it could be solved completely within 2 hours. As a result, the level of knowledge and skills required for exercise solving is lowered, and so is the level of total knowledge that students acquire throughout the course. Another problem with this approach is that every syllabus group has to have a different exercise. We have about 20 syllabus groups and creating 20 similar exercises every week is a serious organizational problem. If the exercises are known in advance, students can make the preparation at home before the syllabus exercises. Therefore, the exercises can be more demanding and the knowledge that a student has to show greater. That leads to a higher level of total knowledge and skills that student will acquire, and also encourages individual work. However, an issue related to disclosing the exercises in advance is plagiarism, which is a serious problem that cannot be solved easily. Owing to the large number of students and the fact that exercises are held by more than one assistant, it is not easy to remember solutions that students make and thus discover plagiarism. The first measure we introduced was the 11th syllabus exercise, which is not disclosed in advance, and is crucial for the final grade. In this way the first ten exercises can be plagiarized, but students cannot get the final grade unless they solve the 11th exercise. As mentioned earlier, the problem with this kind of testing is that every single syllabus group has to have a different exercise. The second measure we introduced was a specialized learning programming interface for writing C++ programs, named Verificator, instead of standard ones. Verificator eliminates a great part of former problems with students' plagiarism and helps our students in their understanding of C++ syntax and adoption of good programming habits. This software is presented in the rest of this paper.

4. Problems Encountered by Our Students in Acquiring Programming Skills

Our students' web-form questionnaire showed some of the most common problems in learning programming (the survey was conducted after the 3rd computer exercise in the course; Table 1).

Some other researches, e.g., Gomes and Mendes (2007) also show that the algorithm for solving a problem is seen as a harder problem than the programming language syntax. Does it mean that the teaching process should be oriented to algorithms only, because

Table 1
Students' problems in learning programming (Radošević *et al.*, 2009)

	% students (2009)
lack of previous knowledge	65%
distracted by other study obligations	58%
algorithm for solving a problem	26%
using the curly brackets	17%
fear of programming	13%
C++ syntax	12%
C++ seems too hard	9%
C++ operators	9%
iterations (for, while, do-while)	9%
C++ instructions	7%
selections (if, switch)	3%
using the semicolon	2%

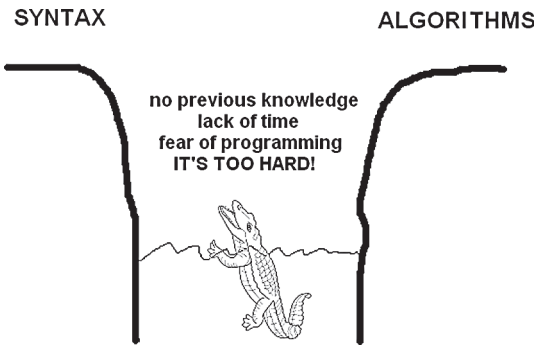

Fig. 1. Difficulties in learning programming.

programming syntax is 'easy'? We cannot by any means agree with that. The multivariate analysis of mutual correlations between students' answers has revealed to us which factors are positively – or negatively – correlated to some of the students' main problems in learning programming (at the level of solving exercises in the Programming 1 course; Table 2):

- fear of programming,
- C++ seems too hard,
- lack of previous knowledge,
- algorithm for solving a problem.

The correlations (although occasional in places and too small to prove the cause/effect association) indicate that problems with programming language syntax are the most common in connection to other problems in learning programming. On the other hand, writing a lot of small programs may be good practice aimed at eliminating the most common syn-

Table 2
Correlations between some of students' main problems in learning programming

Correlation	Problem			
	Fear of programming	C++ seems too hard	Lack of previous knowledge	Algorithm for solving problem
Positive	– while loop (0.26) – lack of previous knowledge (0.25) – do-while loop (0.20) – curly brackets (0.16) – C++ seems too hard (0.15)	– do-while loop (0.21) – C++ instructions (0.19) – while loop (0.17) – C++ operators (0.16) – fear of programming (0.15)	– fear of programming (0.25) – did not pass the Informatics 1 course (0.18) – C++ operators (0.15) – poor knowledge of another programming language (0.15)	– C++ operators (0.27) – C++ instructions (0.20) – distracted by other study obligations (0.16) – fear of programming (0.13) – using the semicolon sign (0.13)
Negative	– want programming in their future job (−0.26) – often write programs of up to 1000 lines (−0.24) – knowledge of computer networks (−0.21) – knowledge of HTML (−0.20) – knowledge of spreadsheets (−0.18)	– knowledge of computer hardware (−0.21) – often write programs of up to 1000 lines (−0.17) – usage of a sound processing tool (−0.17) – knowledge of computer networks (−0.12) – usage of a video processing tool (−0.11)	– often write programs of up to 1000 lines (−0.51) – knowledge of HTML (−0.34) – knowledge of C programming language (−0.32) – usage of more than 1 programming language (−0.30) – writing of programs larger than 1000 lines (−0.24)*	– usage of more than 1 programming language (−0.17) – usage of more than 1 programming language (−0.17) – knowledge of HTML (−0.16) – knowledge of Pascal (−0.13) – usage of a video-processing tool (−0.12) – control term (−0.10)**

* Only 4% of respondents

** Cheat term as a term important for Informatics history (indicates the overvaluation of knowledge)

tax problems, although it is probably not so effective when problems with algorithms are concerned. The problem could be represented by Fig. 1.

To conclude, there is a gap in the interconnection between the programming language syntax and algorithms for solving problems. Our approach to dealing with that problem is to offer our students a lot of practice and a software tool, Verificator, which helps them with the programming language syntax and the adoption of good programming habits.

5. Verificator

Verificator is a kind of learning programming interface which includes a programming editor, debugger, and tools for syntax analysis of C++ program code. It is connected to a standard C++ compiler, like gcc. The main purpose of Verificator is to help our students in

acquiring good programming habits and to avoid the bad ones. During years of teaching experience we have had the opportunity to see many such bad habits developed by our students, e.g.:

- trying to copy program solutions from colleagues,
- learning large program blocks by rote without any understanding,
- writing programs without any syntax and logical check.

The main purpose of Verificator is to push students to make control points during program development, so they cannot continue writing their program until its syntax is correct (i.e., until it can be compiled without errors). This means that all program structures have to be closed and regular according to syntax. Furthermore, programs cannot be imported (e.g., by copy/paste operation or loading from a file) into Verificator, and have to be written manually, with the exception of libraries. Together with a periodical syntax check, it is now much harder to copy programs from colleagues, especially without understanding of program structure and functionality. Programs written in Verificator are personalized because the authors' data are included in form of comments, together with the MD5 checksum which guarantees the program is really written in Verificator. To help our students with the syntax, Verificator includes some tools which enable students to easily find unclosed program structures and parentheses which do not match. The debugger includes marking erroneous code in red and putting a breakpoint into the program.

5.1. *Getting Started with Verificator*

We use Verificator in combination with the DevC++ freeware C++ integrated development environment. After DevC++ is installed, Verificator can be plugged in. Students use Verificator in their computer exercises, but they can also download it for home practice. The work with Verificator starts by filling in the starting form. The data entered in the form are later finished in C++ source code in the form of comments, together with some other metadata, like times and MD5 checksum, e.g.:

```
// MD5:YlUYrh/NBhdmhoh7La2h0Q==
// Verificator
// Program:Example 1
// Description:First learning example.
// Author:Danijel Rado\v{s}evi\'{c}
// Start time:24.6.2009 20:39:19
// Final time:24.6.2009 20:40:44
// IP: ( 340 )
// \#:\#include<iostream>,
// Successful/unsuccessful compilings:2/0
\#include<iostream>
using namespace std;
int main()
{
      int a;
      cout << ''Hello world from Verificator!'' << endl;
      cin >> a;
}
```

The default target for source code is the Desktop folder (to be visible in computer exercises). The file name also contains the data entered in the form (in the example above: *Radosevic_Danijel_Example 1.cpp*).

5.2. *Writing Program Code*

The programming interface contains a semaphore in the form of a traffic light (Fig. 2). Each time a programmer types the semicolon sign or starts a new line, the value on the semaphore increases by one. Values 0–5 are in the green area and mean 'You are free to continue typing'. Values 6–10 (yellow light) mean 'It's time to close the open structure and compile the program'. If the value exceeds 10 (red light on the semaphore), the program cannot be compiled until the programmer reduces the code, after which the semaphore value falls into the yellow or green area. Of course, successful compilation of the program resets the semaphore to zero.

It is important that the comments (marked by '//' or '/* */') are not counted, so students can avoid the red light by commenting on the code and then compile the program. By uncommenting the code it becomes countable, so hiding the code into the comment is not a way to deceive the semaphore.

The idea of a semaphore is to push students to make control points during the development of their programs. So the correctness of the program syntax is checked at different stages of program development. That could not be achieved by sequential retyping of the program, because some structures are too big to be written as a whole before the semaphore goes red. As a result, in the process of writing the program, students need to adhere to its structure.

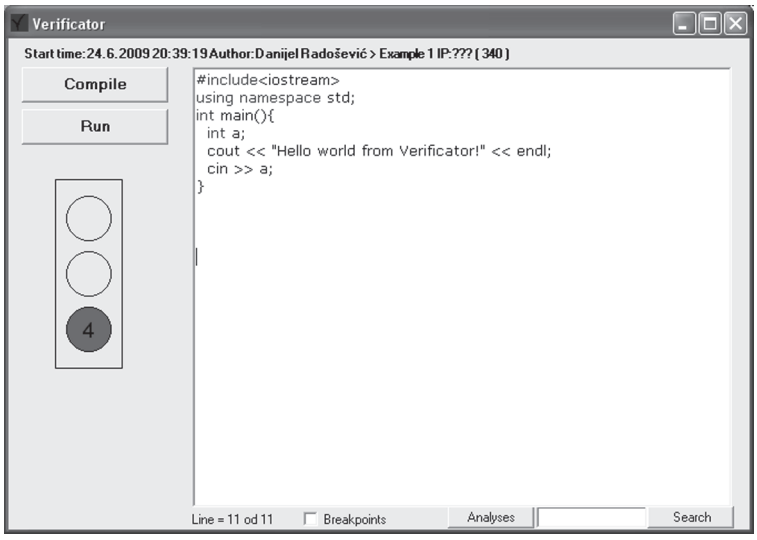

Fig. 2. Programming interface with the semaphore.

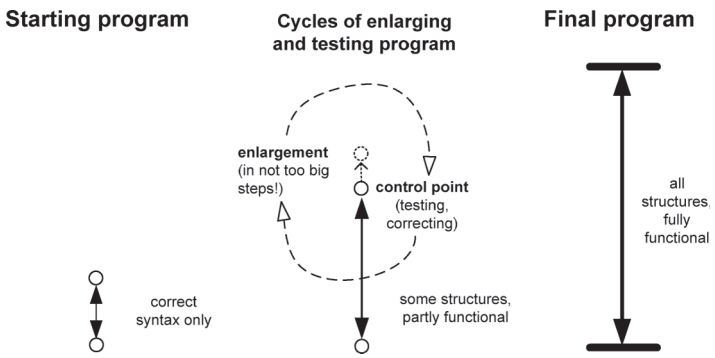

Fig. 3. Program development with control points.

5.2.1. *Control points in writing programs*

Writing a full program before testing it leads to many syntax (and logical) errors and it is hard to help students correct them when they occur in computer exercises. Therefore it is important to follow a logical order in the development of a program. The Verificator semaphore mechanism pushes students to make control points in the development of their programs (Fig. 3).

That means that programming always starts with the easiest possible program, which contains correct syntax only. In C++, that could be the following program:

```
int main(){
       return 0;
}
```

After checking it, the program is ready to be enlarged:

```
#include<iostream>                          //1.
using namespace std;                        //2.
int main(){
       cout « "Hello world!" « endl;        //3.
       return 0;
}
```

It is important to notice that the new program, after adding a few new lines, will hardly contain a large number of errors, except in case where its structure is badly undermined. So, writing a program using Verificator includes two basic steps necessary for acquiring good programming habits:

- step 1: setting the structure and
- step 2: filling the structure with content (instructions and new structures).

In case of the aforementioned program, we can, for example, enlarge it by a loop. Setting the structure includes a loop heading and curly brackets. In some cases, the structure is connected to some data, like the control variable:

```
int i;                          // 1.data connected to structure
for (i=1;i<=10;i++)             // 2.heading
{                               // 3.{ - block beginning
   // structure content (later!)
}                               // 4.} - block end
```

The loop heading in the example depends on a control variable, so it should be declared firstly. Apart from setting the structures before its content, Verificator also pushes its users to take care of the dependencies in the program, e.g.:

- variables cannot be used before their declaration,
- subroutines cannot be called before being set (structure first and content next); this also applies to user defined types like structs, classes and enumerations,
- dynamic structures depend on their allocation in memory,
- using external resources depends on the importing of appropriate libraries.

In addition to helping develop good programming habits, Verificator helps students in learning of programming language syntax by insisting on its correctness at every program development stage.

5.3. *Tools for Syntax Analysis and Debugging*

Verificator includes tools for easier finding of syntax and logical errors in programs. There is a huge emphasis on the teaching approach, because Verificator is aimed at helping our students to find, understand and correct errors in their programs.

5.3.1. *Marking errors in red*

Instead of simply showing the compilers' output, Verificator colors the erroneous code in red. That makes finding errors in program easier.

5.3.2. *Tutor for suggesting solutions*

Error messages received from the compiler are not always intuitive enough, especially for students at the beginners' level of programming, so they often do not understand them, or cannot connect them to real problems in their programs. For example, a missing semicolon could be reported as an error in the next program line, possibly without mentioning any semicolon. Furthermore, is possible that some syntactically correct but logically senseless expressions, e.g.:

```
if (a=b)                    // assignment
```

will not be reported from the compiler as an error, although it is very probable that the programmer's real intention was

```
if (a==b)                   // comparison
```

The tutor appears in the *Errors list* window as an electric bulb each time there is a suggestion to be made (Fig. 4.).

In the example above, the tutor suggests solutions in the following situations:

- forgotten program elements (e.g., basic libraries and namespaces, main),
- unpaired brackets (curly and round) and quotation marks,
- some syntax errors (e.g., usage of semicolon),
- warning about possible problems when the syntax is correct (e.g., usage of '=' instead of '==', variable declarations inside loop headings).

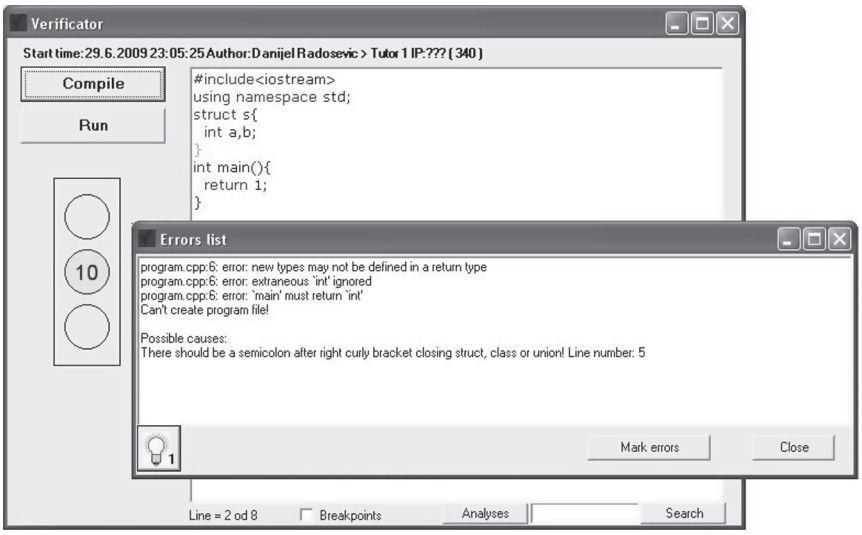

Fig. 4. Typical use of the tutor (the number next to the bulb shows the actual number of suggestions).

5.3.3. *Program analysis*

Problems using curly brackets are very common at the beginners' level of programming. Verificator has a tool for analyzing curly brackets which enables finding opened but unclosed program structures. It's also possible to list all program structures. The information that some of the structures are not referenced can sometimes be useful in finding logical errors.

The debugger enables inserting of breakpoints into program. Breakpoints should be enabled by checking the *Breakpoints* control in Verificator's main window. After that, breakpoints could be inserted into the program in form of comments, e.g.:

//Break: b a

means that the execution of the program will stop to show the current values of variables a and b.

5.4. *Other Verificator Features*

In addition to the features described, Verificator includes some additional features:

- controls for stimulating good programming habits:
  - it is not allowed to put more than one semicolon per line (except *for* loop and *break*),
  - it is not allowed to leave more than five continuous empty rows in a program,
  - periodical warnings (messages) if the number on the semaphore exceeds 18 (which would complicate further writing);
- protection from hacking:
  - queries and answers. Students put queries (in form of a comment) into their programs. The answer (which is displayed as a picture below the semaphore)

depends on the path Verificator is started from, number in the query and binary content of the Verificator program file.
  - time and file size control. It is important that Verificator runs the exact program file made from the source code present in Verificator. For this purpose Verificator remembers the time of creation and size of the compiled program file.

6. Our First Teaching Experiences

Our first teaching experiences can be divided into the following categories:

- effects on the teaching process:
  - it is easier to control students' performance in computer exercises and examinations because Verificator prevents copying programs and writing programs without knowing their structure. So students can use the Internet for finding help,
  - it is easier to help students because Verificator does not allow their errors to become rooted,
  - there is no need for other prohibitions, except for those embedded in Verificator (e.g., prevents copying programs, have to check their programs in the process of writing the code), so students can now use the Internet in doing their exercises for finding help,
  - students have to be better prepared for doing exercises than before;
- positive reactions by students:
  - 'Now it's much easier to find an error and understand the exercise.'
  - 'If it weren't for Verificator, I would never learn programming!'
  - 'If we had used Verificator last year, I'm sure that I would have passed the exam and wouldn't have to enroll the course again!'
  - students' suggestions on how to improve Verificator, e.g., by adding some new features and repairing the bugs;
- negative reactions by students:
  - the limitation of 10 lines before compilation seems too strict,
  - frequent compilation takes too much time,
  - some bugs in starting versions (usually problems with the editor and the semaphore).

At the end of the semester, we conducted the second students' web questionnaire on a population of 55 respondents (the same respondents as in the first questionnaire; unfortunately, it was hard to gather the same number of respondents as in the first questionnaire because the lectures had finished and participation was voluntary). Nevertheless, some correlations obtained might be interesting (Table 3).

It seems that students have solved the majority of their initial problems with basic programming language syntax. However, when they learned some new concepts (e.g.,

Table 3
Correlations between some of students' main problems in learning programming (at the end of the semester)

Correlation	Problem			
	Fear of programming (11% respondents)	C++ seems too hard (11% respondents)	Lack of previous knowledge (53% respondents)	Algorithm for solving problem in majority of exercises (23% respondents)
Positive	– no interest in programming (0.47) – exercises seem too hard (0.41) – exercises seem too hard (0.41) – problem solving algorithm in majority of exercises (0.37) – lack of previous knowledge (0.37)	– usage of semicolon (0.51) – C++ operators (0.47) – programming seems too hard (0.46) – curly brackets (−0.37) – fields (0.36)	– distracted by other study obligations (0.49) – exercises seem too hard (0.42) – problem solving algorithm in majority of exercises (0.42) – programming seems too hard (0.37) – fear of programming (0.33)	– fields (0.45) – functions (0.44) – lack of previous knowledge (0.42) – exercises seem too hard (0.41) – fear of programming (0.37)
Negative	– want programming in their future job (−0.32) – often write programs of up to 1000 lines (−0.22) – curly brackets (−0.17)* – problem solving algorithm in only 1 or 2 exercises (−0.15) – knowledge of HTML (−0.15)	– want programming in their future job (−0.25) – frequent usage of E-learning course materials (−0.23) – often write programs of up to 1000 lines (−0.20) – using the help of colleagues in the same year of study (−0.18)	– often write programs of up to 1000 lines (−0.43) – problem solving algorithm in only 1 or 2 exercises (−0.35) – want programming in their future job (−0.28) – knowledge of C (−0.22) – knowledge of Pascal (−0.22)	– often write programs of up to 1000 lines (−0.28) – usage of other learning materials besides those proposed (−0.18) – knowledge of C (−0.18) – want programming in their future job (−0.16) – programming in Pascal or Java (−0.11)

* Those who have problem using curly brackets have less fear from programming (contrary to the results in the first questionnaire).

structures, functions, etc.), some new problems with syntax appeared (e.g., problems with C++ operators and usage of semicolon). At the end of semester, problems with algorithms are predominantly related with usage of arrays and functions whereas at the beginning of the semester, problems were related with basic syntax. Furthermore, instead of any particular problem with syntax or algorithms, many students reported some "generic" problems, like being distracted by other study obligations or not having enough interest to achieve programming skills. Finally, the "medicine" for majority of problems stays the same: practicing by writing lots of small programs gives the best results.

7. Conclusion and Future Work

Over the years, while teaching programming at the University of Zagreb, Faculty of Organization and Informatics in Varaždin, Croatia, we have faced some problems inherent in the teaching itself as well as those related to the organization of the teaching process. Students' problems in learning programming range from the most common problems, like a lack of foreknowledge and being distracted by other duties in the study to less common, but even harder problems, like the fear of programming and the perception of programming as being too hard. Students occasionally try to find easy ways to pass the exam, which can lead to bad programming habits. Some of these are learning programs or their parts by rote, without any understanding, or trying to write large programs without any syntax and logical test before they are finished. On the other hand, it is hard to help students once their errors are rooted and they are completely discouraged from learning programming.

Our students' web questionnaire and its results show that the majority of problems they encounter in learning programming arise from the gap between the understanding of programming language syntax and problem solving algorithms. Our approach to dealing with that problem is to offer our students a lot of programming practice and our software tool, Verificator, which helps them use the programming language syntax and adopt good programming habits. Verificator prevents students from copying programs from colleagues and pushes them to make control points during the development of their programs with syntax and logical checks. Writing programs using Verificator includes two basic steps, which are necessary for acquiring good programming habits:

- step 1: setting the structure and
- step 2: filling the structure with content (instructions and new structures)

In addition to setting the structure before its contents, Verificator pushes its users to take care of the dependencies in the program, e.g.:

- variables cannot be used before their declaration,
- subroutines cannot be called before being set (structure first and content next). this also applies to user defined types like structs, classes and enumerations,
- dynamic structures depend on their allocation in memory,
- using external resources depends on the importing of appropriate libraries.

Verificator has tools for syntax analysis and debugging, including finding opened but unclosed program structures, finding unreferenced structures and inserting breakpoints into a program.

In our future work, we intend to further develop the tutor for suggesting solutions for common syntax and logical problems, and add some new features like uploading programs to a web server. We also intend to create a similar learning programming interface for Java.

References

Atanasova, G., Hristova, P. (2003). Flowchart interpreter: An environment for software animation representation. In: *Proceedings of the 4th International Conference on Computer Systems and Technologies*. Sofia, Bulgaria, 453–458.

Bergin, J., Stehlik, M., Roberts, J., Pattis, R. (1997). *Karel++, A Gentle Introduction to the Art of Object-Oriented Programming*. Wiley & Sons, New York.

Ben-Ari, M. (1998). Constructivism in computer science education. In: *Proceedings of the 29th SIGCSE Technical Symposium on CS Education*. Atlanta, Georgia, United States, 257–261.

Ben-Bassat Levy, R., Ben-Ari, M., Uronen, P. (2000). An extended experiment with Jeliot 2000. In: *Proceedings of the First International Program Visualization Workshop*. Porvoo, Finland, 131–140.

Brown, M.H., Sedgewick, R. (1984). A system for algorithm animation. *Computer Graphics*, 18(3), 177–186.

Butler, M., Morgan, M. (2007). Learning challenges faced by novice programming students studying high level and low feedback concepts. In: *Proceedings of Ascilite*. Singapore, 99–107.

Cardellini, L. (2002). An Interview with Richard M. Felder. *Journal of Science Education*, 3(2), 62–65.

Carlisle, M., Wilson, T., Humphries, J., Hadfield, M. (2004). RAPTOR: Introducing programming to non-majors with flowcharts. *Journal of Computing Sciences in Colleges*, 19(4), 52–60.

Costelloe, E. (2004). *Teaching Programming The State of the Art*. CRITE Technical Report, Department of Computing, Institute of Technology Tallaght, Dublin, Ireland.
`https://www.cs.tcd.ie/crite/publications/sources/programmingv1.pdf`

Crews, T. (2001). *Using a Flowchart Simulator in an Introductory Programming Course*. Computer Science Teaching Centre Digital Library, Western Kentucky University, USA.
`http://www.citidel.org/bitstream/10117/119/2/Visual.pdf`

Elkner, J. (2004). *The Guido van Robot Programming Language*. Sourceforge.net.
`http://gvr.sourceforge.net/`

Fowler, L., Allen, M., Armarego, J., Mackenzie, J. (2000). Learning styles and CASE tools in software engineering. In: *Proceedings of the 9th Annual Teaching Learning Forum*. Curtin University of Technology, Perth, Australia.
`http://lsn.curtin.edu.au/tlf/tlf2000/fowler.html`

Garner, S. (2003). Learning resources and tools to aid novices learn programming. In: *Proceedings of Informing Science & Information Technology Education Joint Conference*. Pori, Finland, 213–222.

Gomes, A., Mendes, A.J. (2007). Learning to program – difficulties and solutions. In: *Proceedings of the International Conference on Engineering Education*. Coimbra, Portugal.
`http://icee2007.dei.uc.pt/proceedings/papers/411.pdf`

Grissom, S., McNally, M., Naps, T. (2003). Algorithm visualization in CS education: Comparing levels of student engagement. In: *Proceedings of the ACM Symposium on Software Visualization*. San Diego, California, USA, 87–94.

Haajanen, J., Pesonius, M., Sutinen, E., Tarhio, J., Teräsvirta, T., Vanninen, P. (1997). Animation of user algorithms on the Web. In: *Proceedings of 1997 IEEE Symposium on Visual Languages*. Isle of Capri, Italy, 360–367.

Hundhausen, C.D., Douglas, S.A., Stasko, J.D. (2002). A Meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3), 259–290.

Kaila, E., Rajala, T., Laakso, M.-J., Salakoski, T. (2009). Effects, experiences and feedback from studies of a program visualization tool. *Informatics in Education*, 8(1), 17–33.

Kannusmäki, O., Moreno, A., Myller, N., Sutinen, E. (2004). What a novice wants: students using program visualization in distance programming course. In: *Proceedings of the Third Program Visualization Workshop*. Warwick, UK, 126–133.

Kölling, M., Quig, B., Patterson, A., Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Journal of Computer Science Education*, 13(4), 249–268.

Korhonen, A., Malmi, L., Silvasti, P. (2003). TRAKLA2: a framework for automatically assessed visual algorithm simulation exercises. In: *Proceedings of the Third Finnish/Baltic Sea Conference on Computer Science Education*. Koli, Finland, 48–56.

Laakso, M.-J., Salakoski, T., Grandell, L., Qiu, X., Korhonen, A., Malmi, L. (2005). Multi-perspective study of novice learners adopting the visual algorithm simulation exercise system TRAKLA2. *Informatics in Education*, 4(1), 49–68.

Lahtinen, E., Ala-Mutka, K., Järvinen, H-M. (2005). A study of the difficulties of novice programmers. In: *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. Caparica, Portugal, 14–18.

Lattu, M., Tarhio, J., Meisalo V. (2000). How a visualization tool can be used: Evaluating a tool in a research and development project. In: *Proceedings of the 12th Workshop of the Psychology of Programming Interest Group*. Corenza, Italy, 19–32.

Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, 36(4), 119–150.

Lovrenčić, A., Konecki, M., Orehovački, T. (2009). 1957–2007: 50 years of higher order programming languages. *Journal of Information and Organizational Sciences*, 33(1), 79–150.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B.-D., Laxer, C., Thomas, L., Utting, I., Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, 33(4), 125–180.

Milne, I., Rowe, G. (2002). Difficulties in learning and teaching programming – views of students and tutors. *Education and Information Technologies*, 7(1), 55–66.

Mulholland, P. (1998). A principled approach to the evaluation of SV: a case study in Prolog. In: Stasko, J., Domingue, J., Brown, M.H., Price, B.A. (Eds.), *Software Visualization. Programming as a Multimedia Experience*. The MIT Press, 439–451.

Naps, T.L., Rößling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S., Velázquez-Iturbide, J.Á. (2002). Exploring the role of visualization and engagement in computer science education. *ACM SIGCSE Bulletin*, 35(2), 131–152.

Pattis, R.E. (1981). *Karel the Robot*: *A Gentle Introduction to the Art of Programming*. John Wiley & Sons, New York.

Pausch, R., Burnette, T, Capeheart, A.C., Conway, M., Cosgrove, D. DeLine, R., Durbin, J., Gossweiler, R., Koga, S., White, J. (1995). Alice: Rapid prototyping system for virtual reality. *IEEE Computer Graphics and Applications*, 15(3), 8–11.

Radošević, D., Orehovački, T., Lovrenčić, A. (2009). New approaches and tools in teaching programming. In: *Proceedings of Central European Conference on Information and Intelligent Systems* (*Ceciis 2009*). Varaždin, Croatia, 49–57.

Scott, A., Eyres, D., Watkins, M. (2006). Animated flowcharts as an aid to learning programming. In: *Proceedings of the 10th Java in the Internet Curriculum Conference*. North Campus, London Metropolitan University, UK, 12–16.

Spohrer, J.C., Soloway, E. (1986). Novice mistakes: are the folk wisdoms correct? *Communications of the ACM*, 29(7), 624–632.

Stasko, J.T. (1990). Tango: A framework and system for algorithm animation. *IEEE Computer*, 23(9), 27–39.

Stasko, J., Kraemer, E. (1993). A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2), 258–264.

Thomas, L., Ratcliffe, M., Woodbury, J., Jarman, E. (2002). Learning styles and performance in the introductory programming sequence. In: *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*. Cincinnati, Kentucky, USA, 33–42.

TIOBE (2009). *TIOBE Programming Community Index for June 2009*.
`http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

Tudoreanu, M.E. (2003). Designing effective program visualization tools for reducing user's cognitive effort. In: *Proceedings of the 2003 ACM Symposium on Software Visualization*. San Diego, California, 105–114.

D. Radošević, PhD, is an assistant professor at University of Zagreb, Faculty of Organization and Informatics. He teaches at different programming courses. His research focuses on programming languages, generative programming and educational software.

T. Orehovački is a PhD student and teaching assistant at Faculty of Organization and Informatics, University of Zagreb. His research interests include programming education, web technologies, security of web applications and users, knowledge management and generative programming.

A. Lovrenčić, PhD, is an associate professor at University of Zagreb, Faculty of Organization and Informatics. His research is focused on algorithms, programming and data structures.

Verificator: mokomoji priemonė programavimui mokytis

Danijel RADOŠEVIĆ, Tihomir OREHOVAČKI, Alen LOVRENČIĆ

Straipsnyje nagrinėjama mokomoji programavimo sąsaja "Verificator", specialiai skirta universiteto studentams pradedantiems mokytis C++ programavimo kalbos. Mokant programavimo reikia atsižvelgti į tam tikras problemas, susijusias su pačiu mokymu, ir taip pat į mokymo organizavimo eigą. Viena iš didžiausių problemų yra ta, kad studentai, norėdami kuo greičiau atlikti užduotis, susiformuoja kai kuriuos netinkamus programavimo įpročius, pavyzdžiui, bando rašyti programas netikrindami nei sintaksės, nei veiksmų logikos. Įgytus netinkamo programavimo įgūdžius yra gana sunku vėliau ištaisyti. Išnagrinėjus studentų užpildytą internete klausimyną, buvo nustatyta, kad mokantis programavimo, dauguma problemų kyla dėl programavimo kalbos sintaksės ir užduoties algoritmo supratimo. Sukurtoji kompiuterinė mokomoji priemonė "Verificator" padeda studentams įveikti daugelį programavimo klaidų, talkina jiems perprantant programavimo kalbos sintaksės konstrukcijas, sudaro prielaidas geriems programavimo įgūdžiams įgyti.